

# Finding Matches in Unlabelled Streams<sup>\*</sup>

Raphaël Clifford<sup>1</sup>, Markus Jalsenius<sup>1</sup> and Benjamin Sach<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Bristol, U.K.

<sup>2</sup> Department of Computer Science, University of Warwick, U.K.

**Abstract.** We study the problem of finding matches in a stream with unlabelled data. Where the data are not labelled, the only information we have is which items are the same and which differ. A pattern  $P$  of length  $m$  is said to match a substring of the stream  $T$  at position  $i$  if there is an injective (one-to-one) function  $f$  such that  $T[i+j] = f(P[j])$  for all  $0 \leq j < m$ . Such a mapping corresponds to a labelling or relabelling of the symbols in the input and may be distinct for each alignment of the pattern and streaming text. This problem which has also been known under the name parameterised matching has applications from plagiarism detection in computer code to searching within cryptograms. We present both randomised and deterministic solutions. Our deterministic solution requires  $O(|\Sigma| + \rho)$  words of space, where  $|\Sigma|$  is the number of distinct characters in the pattern and  $\rho$  is the parameterised period of the pattern. Our randomised solution improves the space requirements to  $O(|\Sigma| \log m)$  words and is necessarily more sophisticated in its approach. Both algorithms take  $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$  time per new arriving symbol in the worst case. Our randomised algorithm finds all matches with high probability and we show that both space and time requirements are optimal up to logarithmic factors.

## 1 Introduction

We consider the problem of pattern matching in a stream with unlabelled data. In this setting the only information we have about the streaming symbols is which are the same and which differ. The search problem, which is also known as parameterised matching in offline settings, has at its origin the problem of finding duplication and plagiarism in software code although has since found a number of other applications. Since the first introduction of parameterised matching in an algorithmic setting, a great deal of work has gone into its study in both theoretical and practical settings (see e.g. [1, 4–7, 11]). Perhaps the most basic relevant property of parameterised matching is that in an off-line setting, the exact parameterised matching problem can be solved in near linear time using a variant [1] of the classic linear time exact matching algorithm KMP [13].

In our streaming setting, the pattern or query is known in advance and the symbols of the stream arrive one at a time. Our task is to output if there is a match between the pattern and the latest suffix of the stream as soon as a new symbol arrives, where in this case the symbols of the stream are unlabelled. More formally, the pattern  $P$  of length  $m$  is said to match a substring of the stream  $T$  at position  $i$  if there is an injective (one-to-one) function  $f$  such that  $T[i+j] = f(P[j])$  for all  $0 \leq j < m$ . Such a mapping corresponds to a labelling or relabelling of the symbols and may be distinct for each alignment of the pattern and streaming text. The matching problem can be viewed in a number of practical ways but perhaps the simplest is to

---

<sup>\*</sup> This work was supported by EPSRC.

consider the task as that of finding matches in a stream encrypted using different substitution ciphers. To give a small example, the pattern **aba** matches string **xyxyx** at all three alignments but only matches string **xxxyx** at the final alignment.

Our interest is in tackling the parameterised matching problem in a streaming setting using minimal space and with guaranteed worst case running time. The field of pattern matching in a stream took a significant step forwards in 2009 when it was shown to be possible to solve (non-parameterised) exact matching using only  $O(\log m)$  words of space and  $O(\log m)$  time per new stream symbol [15]. This method correctly finds all matches with high probability. The initial approach was subsequently somewhat simplified [9] and then finally improved to run in constant time [8] within the same space requirements. Our results provide the first demonstration that near optimal space and near constant time is achievable for a more challenging problem.

For these previous exact matching methods to work, properties of the periods of strings form a crucial part of their analysis. However, when considering parameterised matching the period of a string is a much less straightforward concept than it is for exact matching. For example, it is no longer true that consecutive matches must either be separated by the period of the pattern or be at least  $m/2$  symbols apart. This property, which does hold for exact matching but does not in the parameterised case, allows for an efficient encoding of the positions of the matches and is crucial to reducing the space requirements of the previous streaming algorithms. Unfortunately parameterised matches can occur at arbitrary positions in the stream, requiring us to find new ways of reducing the storage space used.

This is however not the only challenge that needs to be tackled. A natural way to match two strings under parameterisation is to consider their *predecessor strings*. For a string  $T$ , the predecessor string  $\text{pred}(T)$  is a string of length  $|T|$  with the property that  $\text{pred}(T)[i]$  is the distance counted in numbers of symbols to the previous occurrence of the symbol  $T[i]$  in  $T$ . In other words,  $\text{pred}(T)[i] = d$ , where  $d$  is the smallest positive non-zero value for which  $T[i] = T[i - d]$ . Whenever no such  $d$  exists, we set  $\text{pred}(T)[i] = 0$ . As an example, if  $T = \text{aababcca}$  then  $\text{pred}(T) = 01022014$ . We can now perform parameterised matching offline by only considering predecessor strings using the fundamental fact that two equal length strings  $S$  and  $S'$  have a parameterised match if and only if  $\text{pred}(S) = \text{pred}(S')$  [4]. A plausible approach to solving the streaming problem would now be to translate the problem of parameterised matching in a stream to that of exact matching. This could be achieved by converting both pattern and stream into their corresponding predecessor strings and maintaining fingerprints of a sliding window of the translated input. However, consider the effect on the predecessor string, and hence its fingerprint, of sliding a window in the stream along by one. The leftmost symbol  $x$ , say, will move out of the window and so the predecessor value of the new leftmost occurrence of  $x$  in the new window will need to be set to 0 and the corresponding fingerprint updated. We cannot however afford to store the positions of all characters in even a single window of the text as this will take  $\Theta(m)$  space.

We will show a matching algorithm that solves these problems and others we encounter en route in near constant time per arriving symbol and minimal space. It turns out that achieving the desired space bound without regard to running time is, although by no means trivial, still relatively straightforward compared to the problems encountered tackling both time and space simultaneously. To achieve our final goal the solution we give de-amortises the entire matching process, spreading the work across the time taken by incoming symbols. A number of technical innovations

are now required as a result of this de-amortisation. These will include, amongst others, new uses of fingerprinting method, compressed encodings, a separate deterministic algorithm designed for prefixes of the pattern with small parameterised period as well as a careful scheduling of work to ensure that preliminary answers are computed in time to output matches as soon as they occur.

## 2 Our new results

Our main result is a fast and space efficient algorithm to solve the problem of finding matches in an unlabelled stream.

**Theorem 1.** *There is a randomised algorithm that finds matches in an unlabelled stream and runs in  $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$  time in the worst case per arriving symbol and  $O(|\Sigma| \log m)$  words of space, where  $|\Sigma|$  is the number of distinct symbols in the pattern. The probability that the algorithm outputs correctly at all alignments of an  $n$  length text is at least  $1 - 1/n^c$ , where  $c$  is any constant.*

The running time is therefore near optimal. The full running time is in fact dominated by the complexity of the operations insert, delete and lookup in a dynamic dictionary containing  $|\Sigma|$  distinct elements. Using a worst case variant of exponential search trees [2] we achieve the quoted time complexity but any suitable dictionary data structure can be substituted without change to the overall algorithm.

We also give a separate and somewhat simpler deterministic solution which uses  $O(|\Sigma| + \rho)$  words of space, where  $\rho$  is the parameterised period (q.v. Section 2.1) of the pattern. The time complexity matches that of our randomised solution. We use it as a special case for our main randomised algorithm but it may be of independent interest in cases where the pattern has small parameterised period.

**Theorem 2.** *There is a deterministic algorithm that finds matches in an unlabelled stream and runs in  $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$  time in the worst case per arriving symbol and  $O(\rho + |\Sigma|)$  words of space, where  $\rho$  is the parameterised period of  $P$ .*

To complete the picture we give nearly matching space lower bounds which show that our solutions are optimal to within log factors. The proof is by a relatively straightforward communication complexity argument. In essence one can show that in the randomised case Alice is able to transmit a complete string of length  $\Omega(|\Sigma|)$  bits to Bob using a solution to the matching problem by choosing a suitably crafted pattern and streaming text. Similarly in the deterministic case one can show that she can send a bit string of length  $\Omega(|\Sigma| + \rho)$  bits. The proof is deferred to Appendix A.

**Theorem 3.** *There is a randomised space lower bound for the problem of finding matches in an unlabelled stream of  $\Omega(|\Sigma|)$  bits. There is also a deterministic space lower bound of  $\Omega(|\Sigma| + \rho)$  bits for the same problem.*

### 2.1 Facts, notation and definitions

We use the term *p-match* for a parameterised match and define the *parameterised period* (*p-period*) of a string  $P$  as the smallest  $\rho > 0$  such that  $P[0 \dots (m - 1 - \rho)]$  *p-matches*  $P[\rho \dots m - 1]$ .

We will make extensive use Rabin-Karp style fingerprints of strings which we define as follows. Let  $p > |\Sigma|$  be a prime and choose  $r \in \mathbb{Z}_p$  uniformly at random. For a string  $S$ , the fingerprint  $\phi(S)$  is given by

$$\phi(S) \stackrel{\text{def}}{=} \sum_{k=0}^{|S|-1} S[k]r^k \bmod p.$$

A critical property of the fingerprint function  $\phi$  is that the probability of achieving a false positive, that is the probability that  $P(\phi(S) = \phi(S') \wedge S \neq S') \leq |S|/(p-1)$  (see [12, 15] for proofs). As we assume the RAM model with word size  $\Theta(\log n)$ , where  $n$  is the total length of the stream, we can therefore choose  $p = n^c$  for any constant  $c$ , giving a false positive probability asymptotically no more than  $1/n^{c-1}$ . In particular, as our randomised algorithm will make  $O(n \log n)$  fingerprint comparisons in total, we can instead choose  $p$  so that by the union bound, there are no false positives with the same probability. We assume that all fingerprint arithmetic is performed within  $\mathbb{Z}_p$ , in particular when subtracting one fingerprint from another; an operation we will need to do repeatedly. We will also take advantage of the following properties of fingerprints.

**Fact 1** Splitting: *given  $\phi(S[a \dots c])$  and  $\phi(S[a \dots b])$ , the value of  $\phi(S[b+1 \dots c])$  can be computed in  $O(1)$  time.* Updating: *The fingerprint of a string  $\phi(S[a \dots c])$  can be updated in  $O(1)$  time given some index  $j \in \{a, \dots, c\}$  and a new value  $S[j]$  for that position in the string. We will focus on setting certain values to zero.*

The main algorithm we present will try to match the streaming text with various prefixes of the pattern  $P$ . We define them along with some associated variables in the following definition.

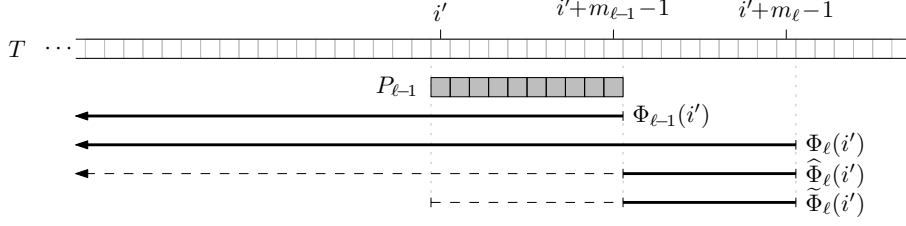
**Definition 1.** Let  $\delta = |\Sigma| \log m$  and let  $P_0$  to be the shortest prefix of  $P$  that has  $p$ -period greater than  $3\delta$ . We define  $s$  prefixes  $P_\ell$  of increasing length so that  $|P_\ell| = 2^\ell |P_0|$  for  $\ell \in \{1, \dots, s-1\}$ , where  $s$  is the largest value such that  $|P_{s-1}| \leq m/2$ . The final prefix  $P_s$  has length  $m - 4\delta$ . For all  $\ell$ , let  $m_\ell \stackrel{\text{def}}{=} |P_\ell|$  denote the length of  $P_\ell$ .

From Definition 1 it follows that  $s \in O(\log m)$ . For our randomised algorithm in Section 3 we will assume that  $m > 14\delta$  to ensure that  $m_\ell - m_{\ell-1} \geq 3\delta$  for  $\ell \in \{1, \dots, s\}$ . If  $m \leq 14\delta$ , or the  $p$ -period of  $P$  is  $3\delta$  or less, we instead apply the deterministic algorithm of Section 4 to solve the problem within the required bounds.

In order to determine if there is a  $p$ -match between the text and a pattern prefix, we will compare fingerprints of the various prefixes of the pattern with fingerprints of the streaming text. We will need three different fingerprint definitions to achieve this (see Fig. 1) as well as a difference fingerprint.

**Definition 2.** For any index  $i'$  and  $\ell$ ,  $\Phi_\ell(i') \stackrel{\text{def}}{=} \phi(\text{pred}(T[0 \dots (i' + m_\ell - 1)]))$ ,  $\widehat{\Phi}_\ell(i') \stackrel{\text{def}}{=} \phi(\text{pred}(T[0 \dots (i' + m_\ell - 1)])[(i' + m_{\ell-1}) \dots (i' + m_\ell - 1)])$  and  $\widetilde{\Phi}_\ell(i') \stackrel{\text{def}}{=} \phi(\text{pred}(T[i' \dots (i' + m_\ell - 1)])[(i' + m_{\ell-1}) \dots (i' + m_\ell - 1)])$ . The fingerprint difference  $\Delta_\ell(i') \stackrel{\text{def}}{=} \widehat{\Phi}_\ell(i') - \widetilde{\Phi}_\ell(i')$ .

In less formal terms, with reference to Fig. 1,  $\Phi_\ell(i')$  is the fingerprint of the predecessor string of the whole text up to index  $i' + m_\ell - 1$ ,  $\widehat{\Phi}_\ell(i')$  is the fingerprint of the  $m_{\ell-1}$  length suffix of the predecessor string of the whole text up to index



**Fig. 1.** The three key fingerprints  $\Phi$ ,  $\hat{\Phi}$  and  $\tilde{\Phi}$ .

$i' + m_{\ell} - 1$ , and  $\tilde{\Phi}_{\ell}(i')$  is similar to  $\hat{\Phi}_{\ell}(i')$ , only that the predecessor string starts at index  $i'$  instead of at the very beginning of the text. Finally the fingerprint difference  $\Delta_{\ell}(i')$  captures the contribution from the positions of the predecessor string of the text that point back beyond position  $i'$ .

As we will see in the next section, all these rather intricate fingerprint definitions make the foundation of our algorithm. We first give a quick example of how our algorithm will take advantage of the properties of fingerprints.

*Example 1.* Assuming that  $P_{\ell-1}$  p-matches the text at position  $i'$  (see Fig. 1 again), our algorithm will work out if the match at  $i'$  can be extended to  $P_{\ell}$  by computing  $\tilde{\Phi}_{\ell}(i')$ . Namely, if  $\tilde{\Phi}_{\ell}(i') = \phi(\text{pred}(P_{\ell})[m_{\ell-1} \dots (m_{\ell} - 1)])$  then by the splitting property of Fact 1,  $P_{\ell}$  p-matches  $T$  at position  $i'$ . Similarly, if  $P_{\ell}$  does not p-match  $T$  at position  $i'$  then with high probability,  $\tilde{\Phi}_{\ell}(i') \neq \phi(\text{pred}(P_{\ell})[m_{\ell-1} \dots (m_{\ell} - 1)])$ . Our algorithm will compute  $\tilde{\Phi}_{\ell}(i')$  by computing  $\hat{\Phi}_{\ell}(i')$  and  $\Delta_{\ell}(i')$  and making use of the updating property of Fact 1.

### 3 The main matching algorithm

Our solution works by finding matches within the stream of the pattern prefixes  $P_0, \dots, P_s$  defined in the previous section, using the observation that if a shorter prefix fails to match at a given position then there is no need to check matches for longer prefixes. Only if all prefixes match at a particular position we check if also the whole of  $P$  matches. To find if a pattern prefix matches, we maintain suitable fingerprints of the streaming text that we update as new symbols arrive and use them to compare to the fingerprints of the pattern prefixes in a similar fashion to Example 1. This overall description also matches that of previous work on exact matching in stream. However, as will become clear, in our case a considerable amount of work is required to simultaneously minimise the space and time requirements.

At any given moment in time our algorithm runs three different processes which we label P1, P2 and P3. Each process takes  $O(1)$  time per arriving symbol once the small-cost alphabet size reduction described in Section 3.4 has been performed. Under this reduction, we may assume that the input alphabet is of the form  $|\Sigma| = \{0, 1, 2 \dots |\Sigma| - 1\}$ . It is under this assumption our algorithm operates. Before describing the processes in more detail, we give a brief overview below. Supporting lemmas for running time, space bounds and correctness are provided in Section 5 with proofs deferred to the appendices.

**Process P1** is responsible for finding matches with prefix  $P_0$  only. To do this it calls a separate deterministic algorithm that we describe in Section 4. When a matching position  $i'$  is found, it will be stored together with the fingerprint  $\Phi_0(i')$

in a queue called  $M_0$  so that process P2 can use it to check if it also matches the prefix  $P_1$ . Total space usage for process P1 is  $O(|\Sigma| \log m)$ .

**Process P2** is responsible for finding matches for all prefixes  $P_\ell$  with  $\ell \geq 1$ , but not for matching the whole pattern which is the responsibility of process P3. Process P2 is randomised and outputs potential matches up to  $3\delta \in O(|\Sigma| \log m)$  symbol arrivals after they occur. The delay is a consequence of our de-amortisation by spreading the work out over arriving symbols.

Process P2 runs two subprocesses labelled P2a and P2b. The first subprocess does bookkeeping of positions of the text whose predecessor values have to be set to zero when taking the fingerprint of substrings of the text. To do this,  $s + 1$  queues are used:  $\mathcal{D}$  and  $D_\ell$  for each  $\ell \in \{1, \dots, s\}$ . Each queue has size  $O(|\Sigma|)$  (which for  $\mathcal{D}$  requires a proof deferred to Section 5).

Subprocess P2b establishes matches for each prefix  $P_\ell$ . Whenever prefix  $P_\ell$  is considered, the subprocess tries to determine if some match with  $P_{\ell-1}$  can be extended to a match  $P_\ell$ . Matches with  $P_{\ell-1}$  are retrieved from a queue  $M_{\ell-1}$  and a newly found match with  $P_\ell$  at position  $i'$  is added to a queue  $M_\ell$ , together with the fingerprint  $\Phi_\ell(i')$ . We show in Section 5 that each such queue can be encoded in a compressed form using only  $O(|\Sigma|)$  space. The subprocess P2b establishes a match at position  $i'$  by first computing the fingerprints  $\hat{\Phi}_\ell(i')$  and  $\Delta_\ell(i')$ . The former is derived from the fingerprint  $\Phi_{\ell-1}(i')$  stored with  $i'$  in  $M_{\ell-1}$ , and the latter is derived from the queue  $D_\ell$ . Together the two fingerprints give  $\tilde{\Phi}_\ell(i')$ , which is used to determine a match (see Example 1). Total space usage for process P2 is  $O(|\Sigma| \log m)$ .

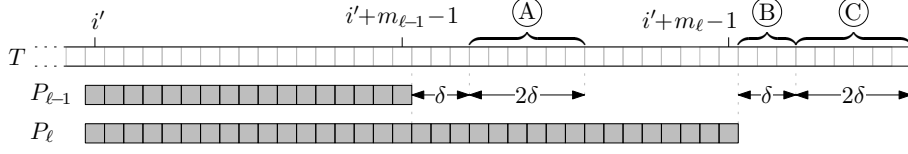
**Process P3** is responsible for finding matches of the whole pattern  $P$  and hence outputting the final answers in constant time. Whenever process P2 reports a match for  $P_s$ , which could occur with up to  $3\delta$  delay, process P3 will naively work out if the match can be extended to the whole pattern  $P$ . This can be done fast enough as there are only  $4\delta$  characters to check over the next  $\delta$  arriving symbols (recall that  $|P_s| = m - 4\delta$ ). Space usage for procedure P3 is  $O(|\Sigma| \log m)$  as  $O(\delta)$  symbols need to be maintained.

### 3.1 Process P1 ( $\ell = 0$ )

Process P1 finds matches in the stream with the pattern prefix  $P_0$ . From the definition of  $P_0$  we have that if we remove the final character from it (giving the string  $P[0 \dots m_0 - 2]$ ) then its p-period is at most  $3\delta$ . Recall that  $\delta = |\Sigma| \log m$ . The p-period of  $P_0$  itself could be much larger. As part of process P1 we run a deterministic pattern matching algorithm on  $P[0 \dots m_0 - 2]$  that returns p-matches with the stream in real-time and (for our pattern) uses  $O(|\Sigma| \log m)$  space. This algorithm, whose space complexity depends on the p-period of the pattern, is described separately in Section 4 and is used here as a stand-alone subroutine.

In order to establish matches with the whole of  $P_0$  we handle the final character separately. If the deterministic subroutine reports a match that ends in  $T[i-1]$ , when  $T[i]$  arrives we have a p-match with  $P_0$  if and only if  $\text{pred}(T)[i] = \text{pred}(P_0)[m_0 - 1]$  (or  $\text{pred}(T)[i] \geq m_0$  if  $\text{pred}(P_0)[m_0 - 1] = 0$ ).

Whenever process P1 finds a match with  $P_0$  at position  $i'$  of the text, the pair  $(i', \Phi_0(i'))$  is added to a (FIFO) queue named  $M_0$ , which is queried by P2.



**Fig. 2.**  $P_{\ell-1}$  and  $P_\ell$  both p-match  $T$  at position  $i'$ . The p-match with  $P_\ell$  is added to  $M_{\ell-1}$  during interval A. Subprocess P2a ensures that by the end of interval B,  $D_\ell$  contains the required elements to compute  $\Delta_\ell(i')$ . Subprocess P2b finds the p-match with  $P_\ell$  during interval C.

### 3.2 Process P2 ( $\ell > 0$ )

Process P2 finds matches in the stream with the pattern prefixes  $P_1, \dots, P_s$ . We split the discussion of its execution into levels and say that level  $\ell$  corresponds to process P2 as it is looking for matches with prefix  $P_\ell$ . Each level is responsible for reporting the matches that occur between its prefix and the different positions in the stream and this information is then used by the subsequent level.

Process P2 computes for each level  $\ell \geq 1$  the fingerprint  $\hat{\Phi}_\ell(i')$  and  $\Delta_\ell(i')$  for each position  $i'$  at which  $P_{\ell-1}$  p-matches the text. Then, as set out in Example 1, if  $\hat{\Phi}_\ell(i') - \Delta_\ell(i')$  (i.e.,  $\tilde{\Phi}_\ell(i')$ ) equals  $\phi(\text{pred}(P_\ell)[m_{\ell-1} \dots (m_\ell - 1)])$ , there is also a match with  $P_\ell$  at  $i'$ . The algorithm will in this case add the pair  $(i', \Phi_\ell(i'))$  to a queue  $M_\ell$ , which is subject to queries by level  $\ell + 1$ .

In order for Process P2 to spend only constant time per arriving symbol, all its work must be scheduled carefully. The preparation of the  $\Delta_\ell$  values takes place as a subprocess we name P2a. Computing the  $\hat{\Phi}_\ell$  values and establishing matches takes place in another subprocess named P2b. The two subprocesses are run in sequence for each arriving symbol. Fig. 2 provides a diagrammatic description of the overall running of P2. We now give details of the two subprocesses.

**Subprocess P2a (preparing the  $\Delta_\ell$  values)** There is a queue  $D_\ell$  associated with each level  $l \geq 1$ . These queues hold positions of the streaming text whose predecessor value points back far enough to be changed to 0 for our purposes.

For each arriving symbol  $T[i]$  we check whether  $\text{pred}(T)[i] > m_0$ . If so, the pair  $(i, \text{pred}(T)[i])$  is added to a (FIFO) queue called  $\mathcal{D}$ , to be dealt with later.

If the subprocess P2a is currently not in the state of processing an element from the queue  $\mathcal{D}$ , it will now remove an element from  $\mathcal{D}$  (unless  $\mathcal{D}$  is empty). Call this element  $(i', \text{pred}(T)[i'])$ . Over the next  $s$  arriving symbols, the subprocess P2a will process this element as follows. For each of the  $s$  levels  $\ell \geq 1$ , if  $\text{pred}(T)[i'] > m_{\ell-1}$ , add  $(i', \text{pred}(T)[i'])$  to the queue  $D_\ell$ . If  $D_\ell$  contains more than  $12|\Sigma|$  elements, discard the oldest element. As we will see shortly, the subprocess P2b will use the information from the  $D_\ell$  queues to work out the  $\Delta_\ell$  values when needed.

**Subprocess P2b (finding matches for all prefixes)** This subprocess schedules the work across the levels in a round robin fashion by only considering level  $\ell = 1 + (i \bmod s)$  when the symbol  $T[i]$  arrives. Potential matches may not be reported by this subprocess until  $O(|\Sigma| \log m)$  arriving symbols after they occur.

The subprocess P2b for level  $\ell$  is always in one of two states. Either it is checking whether a matching position  $i'$  for  $P_{\ell-1}$  can be extended to a match with  $P_\ell$ , or it is idle, waiting to check some reported match with  $P_{\ell-1}$ . In the latter state, level  $\ell$  looks into queue  $M_{\ell-1}$  which contains matches with  $P_{\ell-1}$ . Whenever  $M_{\ell-1}$

becomes or already is non-empty, level  $\ell$  removes an element from  $M_{\ell-1}$ . Call this element  $(i', \Phi_{\ell-1}(i'))$ . When  $i > i' + m_\ell + \delta$  (interval C in Fig. 2), level  $\ell$  will start checking if  $i'$  is also a matching position with  $P_\ell$ . It does so by first computing the fingerprint  $\hat{\Phi}_\ell(i')$ , which from the definition equals  $\Phi_\ell(i') - \Phi_{\ell-1}(i')$ . We can ensure the fingerprint  $\Phi_\ell(i')$  is always available when needed by maintaining a circular buffer of the most recent  $\Theta(|\Sigma| \log m)$  fingerprints of the text.

Over the next at most  $|\Sigma|$  arriving symbols for which P2b is considering level  $\ell$ , the subprocess P2b will compute  $\Delta_\ell(i')$  by stepping through the elements of the queue  $D_\ell$ , aggregating those elements that contribute to the value of  $\Delta_\ell(i')$ . An explicit formula for this calculation is given in Lemma 5 in Section 5. Once this is done, the fingerprint  $\tilde{\Phi}_\ell(i') = \hat{\Phi}_\ell(i') - \Delta_\ell(i')$  can be computed and compared to  $\phi(\text{pred}(P_\ell)[m_{\ell-1} \dots (m_\ell - 1)])$  as in Example 1. If the values are the same, we have a p-match with  $P_\ell$  at position  $i'$  of the text, and the pair  $(i', \Phi_\ell(i'))$  is added to the queue  $M_\ell$ . This occurs before text index  $i' + m_\ell + 3\delta$  arrives i.e. before the end of interval C in Fig. 2. Correctness and Time/Space bounds of P2a and P2b are given in Section 5.

### 3.3 Process P3 (matching the full pattern)

Levels greater than 0 output the position of matches of various prefixes of the pattern with some delay. In order to report matches of the full pattern as soon as a new stream symbol arrives we need one further stage for our matching algorithm. We have that P2 outputs any p-match between  $P_s = P[0 \dots (m - 4\delta - 1)]$  and  $T$  at most  $3\delta$  arrivals after it occurs, i.e. at least  $\delta$  arrivals before a full p-match with  $P$  occurs. Such  $\delta$  length gaps cannot overlap as  $P_s$  has p-period at least  $3\delta$ . This gives us  $\delta$  arrivals to directly compare  $\text{pred}(P)[(m - 4\delta) \dots (m - 1)]$  with  $\text{pred}(T)[(i' + m - 4\delta) \dots (i' + m - 1)]$ , which we obtain by buffering  $\text{pred}(T)$ . By inspection of the definition of predecessor strings, this is sufficient to determine whether a full p-match with  $P$  occurs. The extra space used is dominated by the need to buffer the last  $\Theta(\delta) = \Theta(|\Sigma| \log m)$  values from  $\text{pred}(T)$  and the time is  $O(1)$  per character.

### 3.4 Coping with larger alphabets

The methods we have described require the input alphabet to be in the range  $\{0 \dots |\Sigma| - 1\}$ . In order to handle larger alphabets we map the input alphabet to this range as the stream symbols arrive using a dynamic dictionary. Let  $T'$  denote the stream after this mapping has been applied. We construct a mapping such that for all  $i$ ,  $\text{pred}(T[i - m + 1 \dots i]) = \text{pred}(T'[i - m + 1 \dots i])$  if  $T[i - m + 1 \dots i]$  contains  $|\Sigma|$  or fewer distinct symbols. If  $T[i - m + 1 \dots i]$  contains more than  $|\Sigma|$  distinct symbols then there can not be a p-match. Using a dictionary based on exponential search trees [2], Lemma 1 below gives us the desired result. It is in fact only this mapping process required for larger input alphabets which prevents the algorithms we present from running in real-time.

**Lemma 1.** *There is a mapping that reduces the input alphabet to be in the range  $\{0 \dots |\Sigma| - 1\}$ , runs in  $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$  time per arriving character and uses  $O(|\Sigma|)$  space, where  $|\Sigma|$  is the number of distinct symbols in the pattern. This mapping preserves the locations of the p-matches in the text stream.*

## 4 The deterministic matching algorithm

We now describe a deterministic algorithm for parameterised matching in a stream which requires  $O(\rho + |\Sigma|)$  space, where  $\rho$  is the parameterised period (p-period) of  $P$ . This algorithm is used to search for the prefix  $P_0$  in the full matching algorithm.

**Theorem 4.** *Streaming parameterised matching can be solved deterministically in real-time and  $O(\rho + |\Sigma|)$  space, where  $\rho$  is the p-period of  $P$  and the pattern and text alphabets are of the form  $\Sigma = \{0, 1, 2, \dots, |\Sigma| - 1\}$ .*

We first briefly summarize the overall approach of [1] which our algorithm follows. Whenever some  $T[i]$  arrives, the overall goal is to calculate the largest  $\ell$  such that  $P[0 \dots \ell - 1]$  p-matches  $T[i - \ell + 1 \dots i]$ . A p-match occurs iff  $\ell = m$ . When a new text character  $T[i + 1]$  arrives the algorithm compares  $\text{pred}(T)[i + 1]$  to  $\text{pred}(P)[\ell]$  to determine whether  $P[0 \dots \ell]$  p-matches  $T[i - \ell + 1 \dots i + 1]$  in  $O(1)$  time. If there is a p-match, we continue with the next text character. If there is not, we shift the pattern prefix,  $P[0 \dots \ell - 1]$  along by its p-period  $\rho_\ell$  so that it is aligned with  $T[i - \ell + \rho_\ell + 1 \dots i]$ . This is the next candidate for a p-match. In the original algorithm, the p-periods of all prefixes are stored in an  $m$ -length array called a prefix table.

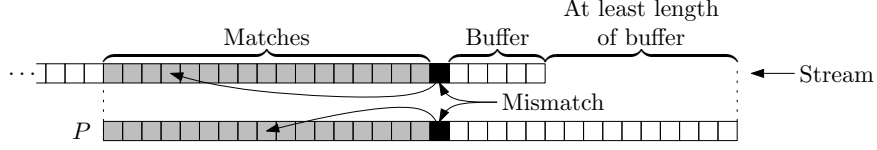
The main hurdle we must tackle is to store both a prefix table suitable for parameterised matching as well as an encoding of the pattern in only  $O(\rho + |\Sigma|)$  space, while still allowing efficient access to both. It is well-known that any string  $P$  can be stored in space proportional to its exact period. In Lemma 2, which follows from Fact 2, we show an analogous result for  $\text{pred}(P)$ .

**Fact 2** *For any  $j \in [\rho]$  there is a constant  $k_j$  such that  $\text{pred}(P)[j + k\rho]$  is zero for  $k < k_j$ , and  $c_j$  for  $k \geq k_j$ , where  $c_j \geq 1$  is a constant that depends on  $j$ .*

**Lemma 2.** *The predecessor string  $\text{pred}(P)$  can be stored in  $O(\rho)$  space, where  $\rho$  is the p-period of  $P$ . Further, for any  $j \in [m]$  we can obtain  $\text{pred}(P)[j]$  from this representation in  $O(1)$  time.*

We now show how to store the parameterised prefix table in only  $O(\rho)$  space, in contrast to  $O(m)$  space which a standard prefix table would require. The p-period  $\rho_\ell$  of  $P[0 \dots \ell]$  is, as a function of  $\ell$ , non-decreasing in  $\ell$ . This property enables us to run-length encode the prefix table and store it as a doubly linked list with at most  $\rho$  elements, hence using only  $O(\rho)$  space. Each element corresponds to an interval of prefix lengths with the same p-period, and the elements are linked together in increasing order (of the common p-period). This representation does not however allow  $O(1)$  time access to the p-period of any prefix, however, for our purposes it will suffice. To accelerate computation we also store a second linked list of the indices of the first occurrences of each symbol in  $P$  in ascending order, i.e. every  $j$  such that  $\text{pred}(P)[j] = 0$ . This uses  $O(|\Sigma|)$  space.

There is a crucial second advantage to compressing the prefix table which is that it allows us to upper bound the number of prefixes of  $P$  we need to inspect when a mismatch occurs. When a mismatch occurs in our algorithm, we repeatedly shift the pattern until a p-match between a text suffix and pattern prefix occurs. Naively it seems that we might have to check many prefixes within the same run. However, by Lemma 3 (which follows from Fact 2) we are assured that if some prefix does not p-match, every prefix in the same run with  $\text{pred}(P)[j] \neq 0$  will also mismatch



**Fig. 3.** A typical state of the deterministic real-time algorithm.

(except possibly the longest). Therefore we can skip inspecting these prefixes. By keeping pointers into both linked lists, it is straightforward to find the next prefix to check in  $O(1)$  time. Whenever we perform a pattern shift we move at least one of the pointers to the left. Therefore the total number of pattern shifts inspected while processing  $T[i]$  is at most  $O(|\Sigma| + \rho)$ . As each pointer only moves to the right by at most one when each  $T[i]$  arrives, an amortised time complexity of  $O(1)$  per character follows. The space usage is  $O(|\Sigma| + \rho)$  as claimed, dominated by the size of the linked lists.

**Lemma 3.** *Let  $j$  be such that  $\rho_j = \rho_{j+1}$ .  $\text{pred}(P)[j - \rho_j] = \{\text{pred}(P)[j], 0\}$ .*

We now briefly discuss how to de-amortise our solution by applying Galil's KMP de-amortisation argument [10]. The main idea is to restrict the algorithm to shift the pattern at most twice when each text character arrives giving a constant time algorithm. If we have not finished processing  $T[i]$  by this point we accept  $T[i + 1]$  but place it on the end of a buffer, output 'no match' and continue processing  $T[i]$ . The key property is that the number of text arrivals until the next p-match occurs is at least the length of the buffer (see Fig. 3). As we shift the pattern up to twice during each arrival we always clear the buffer before (or as) the next p-match occurs. Further, the size of the buffer is always  $O(|\Sigma| + \rho)$ . This follows from the observation above that the number of pattern shifts required to process a single text character is  $O(|\Sigma| + \rho)$ . This therefore establishes Theorem 4. Combining this result with Lemma 1 gives us Theorem 2. The full details are omitted due to space constraints.

## 5 Supporting lemmas: running time, space bounds and correctness

We begin by considering subprocess P2a which maintains the required data structures to efficiently compute the required  $\Delta_\ell$  values. It follows directly from the algorithm description that P2a takes  $O(1)$  time per character and uses  $O(|\Sigma| \log m)$  space to store the queues  $D_\ell$  for all  $\ell$ . Further in Lemma 4, we show that the queue  $\mathcal{D}$  uses only  $O(|\Sigma|)$  space. In simple terms this is because the same symbol can only be inserted into  $\mathcal{D}$  at most every  $\delta = |\Sigma| \log m$  arriving symbols. Coupled with the fact that we remove one element every  $s \in O(\log m)$  arrivals, this ensures that we only ever maintain one element for each symbol in the alphabet.

**Lemma 4.** *After any text arrival, for any  $\sigma \in \Sigma$  there is at most one element  $(i', \text{pred}(T)[i'])$  in  $\mathcal{D}$  with  $T[i'] = \sigma$  and therefore  $\mathcal{D}$  requires only  $O(|\Sigma|)$  space.*

We now turn our attention to subprocess P2b which finds matches with each of the prefixes  $P_1, \dots, P_s$ . We begin in Lemma 5 by giving an explicit expression for  $\Delta_\ell(i')$  in terms of  $D_\ell$ . The lemma follows from the fact that any text index,  $k$ ,

which contributes to some  $\Delta_\ell(i')$  has  $\text{pred}(T)[k] > m_{\ell-1}$  and that there can only be  $O(|\Sigma|)$  such indices in any  $\Theta(m_{\ell-1})$  length window of  $T$ .

**Lemma 5.** *When  $T[i]$  arrives, for any  $i'$  with  $i - 3\delta < i' + m_\ell \leq i - \delta$ ,*

$$\Delta_\ell(i') = \sum_{k=i'+m_{\ell-1}}^{i'+m_\ell-1} d(k) \cdot r^k \bmod p \text{ where } d(k) = \begin{cases} h & \text{if } (k, h) \in D_\ell \text{ and } h > k - i' \\ 0 & \text{otherwise.} \end{cases}$$

Consider the point in Subprocess P2b when computation on  $(i', \Phi_{\ell-1}(i'))$  in  $M_{\ell-1}$  has just begun. This occurs when the first index  $i > i' + m_\ell + \delta$  arrives such that  $\ell = 1 + (i \bmod s)$ . To compute  $\tilde{\Phi}_\ell(i')$ , P2b requires  $\Phi_{\ell-1}(i')$  and  $\Phi_\ell(i')$  (which are readily available) and  $\Delta_\ell(i')$ . To compute  $\Delta_\ell(i')$ , P2b inspects a constant number of elements of  $D_\ell$  once in every  $s$  text arrivals. Therefore as  $|D_\ell| \leq 12|\Sigma|$ , all elements can be inspected before index  $i' + m_\ell + 2\delta$  arrives, as required. From Lemma 5 we have that  $\Delta_\ell(i')$  and hence  $\tilde{\Phi}_\ell(i')$  is computed correctly. Further, as  $i' + m_\ell + 2\delta \leq i' + m_{\ell+1}$ , we have that P2b adds  $(i', \Phi_\ell(i'))$  to the queue  $M_\ell$  (if a match occurs) before it is needed by level  $\ell + 1$ . As the p-period of  $P_\ell$  is more than  $3\delta$ , any two matches in  $M_\ell$  are at least  $3\delta$  positions apart so there is no risk that subprocess P2b will overlook an element of  $M_\ell$  while processing another.

We now examine the space usage for the  $M_\ell$  queues. For all  $\ell$ , whenever a match with  $P_\ell$  at some position  $i'$  of the text is found, the pair  $(i', \Phi_\ell(i'))$  is inserted into the queue  $M_\ell$ . This pair will stay in  $M_\ell$  until it is removed by the subprocess P2b for the purpose of determining whether  $i'$  is also a matching position with  $P_{\ell+1}$ . Despite the delays by which our algorithm perform certain actions, it should not be too difficult to verify that the pair is always inserted into  $M_\ell$  before it is needed. However, it could take up to  $2m_\ell + 2\delta$  text symbol arrivals until the pair is removed. In a windows of this length there could be up to  $\Theta(m_\ell/\rho_\ell)$  matching positions with  $P_\ell$ , where  $\rho_\ell$  is the p-period of  $P_\ell$ . When  $\rho_\ell$  is relatively small, explicitly storing this number of pairs would require much more than  $O(|\Sigma|)$  space. As we will see in the next lemma, there is a succinct data structure that allows us to store every pair in  $M_\ell$  in only  $O(|\Sigma|)$  space.

**Lemma 6.** *For every  $\ell$ , there is a data structure for  $M_\ell$  that uses only  $O(|\Sigma|)$  space. Both retrieving and inserting a pair take  $O(1)$  time.*

The proof of this lemma is arguably the most involved part of the paper, taking advantage of the properties of parameterised matches. Unlike exact matching, parameterised matches that are not too far apart can occur at an arbitrary distance from each other, prohibiting a space efficient representation of their locations. Fortunately however, it turns out that only  $O(|\Sigma|)$  matches show this arbitrary behaviour. Over a certain window of frequent matches, only the first  $O(|\Sigma|)$  matches appear random, after which subsequent matches are evenly spread out with the same distance apart. We can therefore store the first matches explicitly and encode the other matches with an arithmetic progression. Further, the fingerprints that are stored together with the matching positions of the arithmetic progression submit to a regular pattern that we can represent succinctly in  $O(|\Sigma|)$  space.

We have shown that if all the fingerprint comparisons are correct (free from false positives) then our algorithm outputs exactly the locations where each  $P_\ell$  p-matches  $T$  at most  $3\delta$  characters after they occur. As stated in Section 2.1, the probability of a least one false positive occurring can be made as small as  $1/n^c$  for any constant  $c$ . The overall space complexity is  $O(|\Sigma| \log m)$  and the time complexity is  $O(1)$  per

character (after the alphabet reduction) as claimed. Coupled with the discussions in Sections 3.3 and regarding matching the full pattern and Section 3.4 regarding the alphabet reduction, this completes the proof of Theorem 1.

## References

- [1] A. Amir, M. Farach, and S. Muthukrishnan. “Alphabet dependence in parameterized matching”. In: *Information Processing Letters* 49.3 (1994), pp. 111 – 115.
- [2] A. A. Andersson and M. Thorup. “Tight(er) worst-case bounds on dynamic searching and priority queues”. In: *STOC ’00: Proc. 32<sup>nd</sup> Annual ACM Symp. Theory of Computing*. 2000, pp. 335–342.
- [3] Y. Arbitman, M. Naor, and G. Segev. “De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results”. In: *ICALP ’09: Proc. 26<sup>th</sup> International Colloquium on Automata, Languages and Programming*. 2009, pp. 107–118.
- [4] B. S. Baker. “A theory of parameterized pattern matching: algorithms and applications”. In: *STOC ’93: Proc. 25<sup>th</sup> Annual ACM Symp. Theory of Computing*. 1993, pp. 71–80.
- [5] B. S. Baker. “Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance”. In: *SIAM Journal on Computing* 26.5 (1997), pp. 1343–1362.
- [6] B. S. Baker. “Parameterized Pattern Matching: Algorithms and Applications”. In: *Journal of Computer System Sciences* 52.1 (1996), pp. 28 –42.
- [7] B. S. Baker. “Parameterized Pattern Matching by Boyer-Moore-Type Algorithms”. In: *SODA ’95: Proc. 6<sup>th</sup> ACM-SIAM Symp. on Discrete Algorithms*. 1995, pp. 541–550.
- [8] D. Breslauer and Z. Galil. “Real-Time Streaming String-Matching”. In: *CPM ’11: Proc. 22<sup>nd</sup> Annual Symp. on Combinatorial Pattern Matching*. 2011, pp. 162–172.
- [9] F. Ergun, H. Jowhari, and M. Sağlam. “Periodicity in streams”. In: *RANDOM ’10: Proc. 14<sup>th</sup> Intl. Workshop on Randomization and Computation*. 2010, pp. 545–559.
- [10] Z. Galil. “String Matching in Real Time.” In: *Journal of the ACM* 28.1 (1981), pp. 134–149.
- [11] C. Hazay, M. Lewenstein, and D. Sokol. “Approximate parameterized matching”. In: *ACM Trans. Algorithms* 3.3 (2007).
- [12] R. M. Karp and M. O. Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM Journal of Research and Development* 31.2 (1987), pp. 249 –260.
- [13] D. E. Knuth, J. H. Morris, and V. B. Pratt. “Fast pattern matching in strings”. In: *SIAM Journal on Computing* 6 (1977), pp. 323–350.
- [14] R. Pagh and F. F. Rodler. “Cuckoo hashing”. In: *Journal of Algorithms*. 51.2 (2004), pp. 122–144.
- [15] B. Porat and E. Porat. “Exact And Approximate Pattern Matching In The Streaming Model”. In: *FOCS ’09: Proc. 50<sup>th</sup> Annual Symp. Foundations of Computer Science*. 2009, pp. 315–323.

## A Proof of Theorem 3

*Proof (Proof of Theorem 3).* Consider first a pattern where all symbols are distinct, e.g.  $P = 123456$ . Now let us assume Alice would like to send a bit-string to Bob. She can encode the bit-string as an instance of the parameterised matching problem in the following way. As an example, assume the bit-string is 01011. She first creates the first half of a text stream **aBcDE** where we choose capitals to correspond to 1 and lower case symbols to correspond to 0 from the original bit-string. She starts the matching algorithm and runs it until the pattern and the first half of the text have been processed and then sends a snapshot of the memory to Bob. Bob then continues with the second half of the text which is fixed to be the sorted lower case symbols, in this case **abcde**. Where Bob finds a parameterised match he outputs a 1 and where he does not, he outputs a 0. Thus Alice’s bit-string is reproduced by Bob. In general, if we restrict the alphabet size of the pattern to be  $|\Sigma|$  then Alice can similarly encode a bit-string of length  $|\Sigma| - 1$ , and successfully transmit it to Bob, giving us an  $\Omega(|\Sigma|)$  bit lower bound on the space requirements of any streaming algorithm. If randomisation is not allowed, the lower bound increases to  $\Omega(|\Sigma| + \rho)$  bits of space. Here  $\rho$  is the parameterised period of the pattern. This bound follows by a similar argument by devising a one-to-one encoding of bit-strings of length  $\Theta(\rho)$  into  $P[0 \dots \rho - 1]$ . The key difference is that with a deterministic algorithm, Bob can enumerate all possible  $m$ -length texts to recover Alice’s bit-string from  $P$ .

## B Proofs omitted from Section 3.4

*Proof (Proof of Lemma 1).* Let  $|\Sigma|$  be the number of distinct symbols in  $P$ . Let  $\Sigma_T$  denote the text alphabet of the (unfiltered) stream. Let the strings  $S$  and  $S_{\text{filt}}$  denote the last  $m$  characters of the unfiltered and filtered stream, respectively. Let  $\Sigma_{\text{last}} \subseteq \Sigma_T$  denote the up to  $|\Sigma| + 1$  last distinct symbols in  $S$ , hence  $|\Sigma_{\text{last}}|$  is never more than  $|\Sigma| + 1$ . Let  $\mathcal{T}$  be a dynamic perfect hash function on  $\Sigma_{\text{last}}$  such that a symbol in  $\Sigma_T$  can be looked up, deleted and added to  $\mathcal{T}$  in expected  $O(1)$  time [3, 14]. Every symbol that arrives in the stream is associated with its “arrival time”, which is an integer that increases by one for every new symbol arriving in the stream. Let  $\mathcal{L}$  be an ordered list of the symbols in  $\Sigma_{\text{last}}$  (together with their most recent arrival time) such that  $\mathcal{L}$  is ordered according to the most recent arrival time. For example,

$$\mathcal{L} = (\mathbf{d}, 25), (\mathbf{b}, 33), (\mathbf{g}, 58), (\mathbf{e}, 102) \tag{1}$$

means that the symbols **b**, **d**, **e** and **g** are the last four distinct symbols that appear in  $S$  (for this example,  $|\Sigma| + 1 \geq 4$ ), where the last **e** arrived at time 102, the last **g** arrived at time 58, and so on.

By using appropriate pointers between elements of the hash table  $\mathcal{T}$  and elements of  $\mathcal{L}$  (which could be implemented as a linked list), we can maintain  $\mathcal{T}$  and  $\mathcal{L}$  in time  $O(1)$  per arriving symbol. To see this, take the example in Equation (1) and consider the arrival of a new symbol  $x$  at time 103 (following the last symbol **e**). First we look up  $x$  in  $\mathcal{T}$  and if  $x$  already exists in  $\Sigma_{\text{last}}$ , move it to the right end of  $\mathcal{L}$  by deleting and inserting where needed and update the element to  $(x, 103)$ . Also check that the leftmost element of  $\mathcal{L}$  is not a symbol that has been pushed outside of  $S$  when  $x$  arrived. We use its arrival time to determine this and remove the last element accordingly. If the arriving symbol  $x$  does not already exist in  $\Sigma_{\text{last}}$ , then we add  $(x, 103)$  to the right end of  $\mathcal{L}$ . To ensure that  $\mathcal{L}$  does not contain more than

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
pred(P) =			0					0					4					3		...
						0	1	2	3	4	5	6	7	8	9	10	11	12	13	
pred(P) =				0						0								4		...

**Fig. 4.** Here we have supposed that the parameterised period of  $P$  is 5. However, the predecessor string shown contradicts this.

$|\Sigma| + 1$  elements, we remove the leftmost element of  $\mathcal{L}$  if necessary. We also remove the leftmost symbol if it has been pushed outside of  $S$ . The hash table  $\mathcal{T}$  is of course updated accordingly as well.

Let  $\Sigma_{\text{filt}} = \{0, \dots, |\Sigma|\}$  denote the symbols outputted by the filter. We augment the elements of  $\mathcal{L}$  to maintain a mapping  $\mathcal{M}$  from the symbols in  $\Sigma_{\text{last}}$  to distinct symbols in  $\Sigma_{\text{filt}}$  as follows. Whenever a new symbol is added to  $\Sigma_{\text{last}}$ , map it to an unused symbol in  $\Sigma_{\text{filt}}$ . If no such symbol exists, then use the symbol that is associated with the symbol of  $\Sigma_{\text{last}}$  that is to be removed from  $\Sigma_{\text{last}}$  (note that  $|\Sigma_{\text{last}}| \leq |\Sigma_{\text{filt}}|$ ). The mapping  $\mathcal{M}$  specifies the filtered stream: when a symbol  $x$  arrives, the filter outputs  $\mathcal{M}(x)$ . Finding  $\mathcal{M}(x)$  and updating  $\mathcal{T}$  is done in  $O(1)$  time per arriving character, and both the tree  $\mathcal{T}$  and the list  $\mathcal{L}$  can be stored in  $O(|\Sigma|)$  space.

It remains to show that the filtered stream does not induce any false matches or miss a potential match. Suppose first that the number of distinct symbols in  $S$  is  $|\Sigma|$  or fewer. That is,  $\Sigma_{\text{last}}$  contains all distinct symbols in  $S$ . Every symbol  $x$  in  $S$  has been replaced by a unique symbol in  $\Sigma_{\text{filt}}$  and the construction of the filter ensures that the mapping is one-to-one. Thus,  $\text{pred}(S_{\text{filt}}) = \text{pred}(S)$ . Suppose second that the number of distinct symbols in  $S$  is  $|\Sigma| + 1$  or more. That is,  $|\Sigma_{\text{last}}| = |\Sigma| + 1$  and therefore  $S_{\text{filt}}$  contains  $|\Sigma| + 1$  distinct symbols. Thus,  $\text{pred}(S_{\text{filt}})$  cannot equal  $\text{pred}(P)$ .

## C Proofs omitted from Section 4

We first prove two supporting lemmas.

*Proof (Proof of Fact 2).* Let  $\rho$  be the period of  $P$ . We prove the lemma by contradiction. Suppose, for some  $j$  and  $k$ , that  $i = j + k\rho$  is a position such that  $\text{pred}(P)[i] = c \geq 1$  and  $\text{pred}(P)[i + \rho] = c' \neq c$ . Consider Figure 4 for a concrete example, where  $\rho = 5$ ,  $i = 12$ ,  $\text{pred}(P)[12] = c = 4$  and  $\text{pred}(P)[12 + 5] = c' = 3$ .

Since  $\rho$  is a period of  $P$ , we have that

$$\text{pred}(P[\rho \dots m - 1]) = \text{pred}(P[0 \dots m - 1 - \rho]).$$

Consider the alignment of positions  $i + \rho$  and  $i$  (positions 17 and 12 in Figure 4). We have that  $\text{pred}(P[\rho \dots m - 1])[i]$  is either  $c'$  or 0. In either case, it is certainly not  $\text{pred}(P[0 \dots m - 1 - \rho])[i]$  which is  $c$ . Thus,  $\rho$  cannot be a period of  $P$ .

*Proof (Proof of Lemma 2).* By Fact 2 we can encode  $\text{pred}(P)$  by storing the two values  $k_j$  and  $c_j$  for each  $j \in [\rho]$ . This takes  $O(\rho)$  space. The value  $\text{pred}(P)[i]$  is 0 if  $i < k_{(i \bmod \rho)}$ , otherwise it is  $c_{(i \bmod \rho)}$ .

## D Proofs omitted from Section 5

*Proof (Proof of Lemma 4).* Proof by induction. After  $T[0]$  is processed,  $\mathcal{D}$  is empty, providing a base case. Let  $T[i] = \sigma$  be the most recently arrived character. We assume by the strong inductive hypothesis that for all  $i' < i$ ,  $\mathcal{D}$  contains at most one occurrence of each symbol. If  $\text{pred}(T)[i] < m_0$  then  $i$  is not added to  $\mathcal{D}$  and we are done. Therefore, we assume that  $\text{pred}(T)[i] > m_0$ . By the inductive hypothesis, after  $T[i]$  arrives,  $\mathcal{D}$  contains at most one occurrence of each symbol except possibly  $\sigma$ . We now show that  $\sigma$  occurs in  $\mathcal{D}$  exactly once after  $T[i] = \sigma$  arrives.

Consider the state of  $\mathcal{D}$  after  $T[i - \text{pred}(T)[i]]$ , the last  $\sigma$ , arrived. At that point,  $\mathcal{D}$  contained at most  $|\Sigma|$  entries. After  $s$  arrivals after  $T[i - \text{pred}(T)[i]]$ , the occurrence of  $\sigma$  will have moved up one position in  $\mathcal{D}$ . After  $|\Sigma|s$  positions, the  $\sigma$  will have been removed from the queue. However,  $m_0$  must be at least the p-period of  $P[0 \dots m_0 - 1]$  which is greater than  $3\delta$ . Therefore, as  $m_0 > 3\delta > |\Sigma|s$  it is removed before  $T[i] = \sigma$  arrives.

*Proof (Sketch proof of Lemma 5).* By definition we have that  $\Delta_\ell(i') = \widehat{\Phi}_\ell(i') - \widetilde{\Phi}_\ell(i')$ . By rearranging the definitions of  $\widehat{\Phi}_\ell(i')$  and  $\widetilde{\Phi}_\ell(i')$  it follows that,  $\Delta_\ell(i') = \sum_{j=m_{\ell-1}}^{m_\ell-1} d'(i'+j) \cdot r^{i'+j} \bmod p$ . Here  $d'(i'+j) = \text{pred}(T)[i'+j]$  if  $\text{pred}(T)[i'+j] > j$  and 0 otherwise. As  $j \geq m_{\ell-1}$  every such  $(i'+j)$  will be inserted into  $D_\ell$ . The claim follows by observing that by the algorithm description these indices will be present in  $D_\ell$  while  $i - 3\delta < i' + m_\ell \leq i - \delta$ .

### D.1 Storing $M_\ell$ for all $\ell$

We now introduce a some notation, after which we give a few lemmas that will be useful for the proof of Lemma 12 which in turn will allow us to prove Lemma 6.

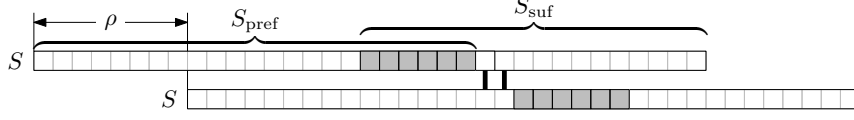
We write  $P \stackrel{\text{p}}{=} T$  to denote a parameterised match between  $P$  and  $T$ . An alphabet can be augmented with the special symbol “ $\star$ ” which is used to represent a so called *wildcard* symbol or a “don’t care” symbol. In terms of matching, the symbol “ $\star$ ” is allowed to match any other symbol of the alphabet without causing a mismatch. For example, the string **ab $\star$ a** matches both **abba** and **abca**. In addition to the predecessor string  $\text{pred}(T)$  of  $T$ , we define the *wildcard predecessor string* of  $T$ , denoted  $\text{pred}^*(T)$ , to be identical to  $\text{pred}(T)$  only with the zeros of  $\text{pred}(T)$  being replaced by the symbol “ $\star$ ”. Thus, if  $\text{pred}(T) = 0102201$  then  $\text{pred}^*(T) = \star 1 \star 22 \star 1$ .

**Lemma 7.** *Suppose  $S$  is a string and  $\text{pred}(S)[i \dots i + m] = \text{pred}(S)[j \dots j + m]$  for some  $i, j$  and  $m$ . Then  $\text{pred}(S[i \dots i + m]) = \text{pred}(S[j \dots j + m])$ .*

*Proof.* The statement of the lemma follows directly from the observation that  $\text{pred}(S[i \dots i + m])$  is uniquely obtained from  $\text{pred}(S)[i \dots i + m]$  by setting every position  $d$  for which  $(\text{pred}(S)[i \dots i + m])[d] > d$  to zero.

**Lemma 8.** *Suppose  $T$  is a string of length  $n$  and  $P$  is a string of length  $m \leq n$ . The string  $P$  parameterise matches  $T$  at position  $i$  if and only if  $\text{pred}^*(P) = \text{pred}(T)[i \dots i + m - 1]$  and the number of distinct symbols in  $T[i \dots i + m - 1]$  equals the number of distinct symbols in  $P$ .*

*Proof.* Let  $R = \text{pred}(T)[i \dots i + m - 1]$ . We first show that if  $P \stackrel{\text{p}}{=} T[i \dots i + m - 1]$  then  $\text{pred}^*(P) = R$ . Let  $R' = \text{pred}(T[i \dots i + m - 1])$  and note that  $R$  and  $R'$  differ only at positions  $j$  where  $R[j] > j$  and  $R'[j] = 0$  (the previous occurrence of symbol  $T[i + j]$  is before index  $i$ ). Since  $P \stackrel{\text{p}}{=} T[i \dots i + m - 1]$ , we have that  $\text{pred}(P) = R'$ ,



**Fig. 5.** Diagram supporting the proof of Lemma 10.

hence  $\text{pred}^*(P) = R$ . Further, since  $P \stackrel{p}{=} T[i \dots i + m - 1]$ , the number of distinct symbols in  $T[i \dots i + m - 1]$  must equal the number of distinct symbols in  $P$ .

Now suppose that  $\text{pred}^*(P) = R$  and suppose  $n_{\text{syms}}$  is the number of distinct symbols in  $T[i \dots i + m - 1]$ , which equals the number of distinct symbols in  $P$ . We will show that  $P \stackrel{p}{=} T[i \dots i + m - 1]$ . Since  $\text{pred}^*(P) = R$  we also have that  $\text{pred}^*(P) = R'$ . The number of zeros in  $R'$  is  $n_{\text{syms}}$  and the number of wildcards “ $\star$ ” in  $\text{pred}^*(P)$  is also  $n_{\text{syms}}$ . Hence  $\text{pred}(P) = R'$ , which implies that  $P \stackrel{p}{=} T[i \dots i + m - 1]$ .

**Lemma 9.** *Suppose  $S$  is a string of length  $n$  and  $\alpha$  is a number such that the prefix  $S[0 \dots \alpha]$  contains all distinct symbols in  $S$ . If  $\rho$  is a parameterised period of  $S$  then  $\rho$  is an exact period of  $\text{pred}(S)[\alpha + 1 \dots n - 1]$  (not necessarily the shortest exact period).*

*Proof.* Since  $\rho$  is a period of  $S$ , we have that  $S[0 \dots n - 1 - \rho]$  parameterise matches  $S$  at position  $\rho$ . From Lemma 8 we have that

$$\text{pred}^*(S[0 \dots n - 1 - \rho]) = \text{pred}(S)[\rho \dots n - 1],$$

where the left-hand side is identical to  $\text{pred}^*(S)[0 \dots n - 1 - \rho]$ . The equation must hold for any corresponding substrings of the left-hand side and right-hand side. In particular, if we ignore the first  $\alpha + 1$  characters, we have

$$\text{pred}^*(S)[\alpha + 1 \dots n - 1 - \rho] = \text{pred}(S)[\rho + \alpha + 1 \dots n - 1].$$

The left-hand side does not contain any wildcards since all distinct symbols in  $S$  occur in  $S[0 \dots \alpha]$ . We can therefore rewrite the equation above as

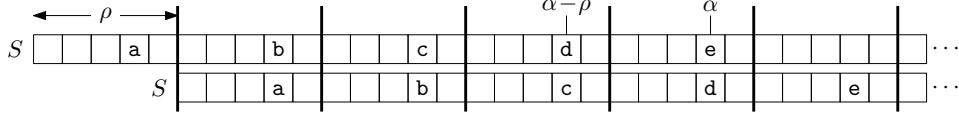
$$\text{pred}(S)[\alpha + 1 \dots n - 1 - \rho] = \text{pred}(S)[\rho + \alpha + 1 \dots n - 1].$$

From the definition of period, it follows that  $\rho$  is an exact period of  $\text{pred}(S)[\alpha + 1 \dots n - 1]$ .

**Lemma 10.** *Suppose  $S$  is a string, and  $S_{\text{pref}}$  is a prefix of  $S$  and  $S_{\text{suf}}$  is a suffix of  $S$ . If  $\rho$  is a period of both  $S_{\text{pref}}$  and  $S_{\text{suf}}$ , and the length of the overlap of  $S_{\text{pref}}$  and  $S_{\text{suf}}$  is at least  $\rho$ , then  $\rho$  is a period of  $S$ .*

*Proof.* Consider  $S$  aligned with itself shifted  $\rho$  steps in Figure 5. The shaded area is the overlap of  $S_{\text{pref}}$  and  $S_{\text{suf}}$ . Since  $\rho$  is a period of both  $S_{\text{pref}}$  and  $S_{\text{suf}}$ , there is a match at every position of the alignment, with the possible exception of the positions marked by the thick vertical line segments. If the length of the overlap is at least  $\rho$ , no such positions exist.

**Lemma 11.** *Suppose  $S$  is a string of length  $n$  over the alphabet  $\Sigma$ . Let  $\alpha$  be the smallest number such that all distinct symbols in  $S$  occur in the prefix  $S[0 \dots \alpha]$ . Then the parameterised period of  $S$  is at least  $\alpha/|\Sigma|$ .*



**Fig. 6.** Diagram supporting the proof of Lemma 11.

*Proof.* Let  $\rho$  be the parameterised period of  $S$ . From the definition of  $\alpha$ , we have that the symbol  $S[\alpha]$  does not occur before position  $\alpha$ . Consider the diagram in Figure 6, where  $S[\alpha] = e$ . Since  $S$  parameterise matches itself when shifted  $\rho$  steps, we have that the symbol  $S[\alpha - \rho]$  cannot occur before position  $\alpha - \rho$  as this would require  $S[\alpha]$  to occur before position  $\alpha$ . We repeat this argument by shifting again, and conclude that the number of symbols  $|\Sigma| \geq \alpha/\rho$ . Thus,  $\rho \geq \alpha/|\Sigma|$ .

Recall that an *arithmetic progression* is a sequence of numbers such that the difference between any two successive numbers in the sequence is constant. We can specify an arithmetic progression by its start number, the difference between successive numbers and the length of the sequence. For notational convenience, we think of an arithmetic progression as a set of numbers (for which there is a very succinct representation). In the next lemma we will see that the positions at which a string  $P$  of length  $m$  parameterise matches a longer string  $T$  of length  $3m/2$  can be stored in small memory: either a matching position belongs to an arithmetic progression or it is one of very few positions that can be listed explicitly. We will now show how this fact can be used to store all the required fingerprints in  $O(|\Sigma|)$  space (per level).

**Lemma 12.** *Let  $X \subseteq \{n - 3m/2, \dots, n - 1\}$  be the set of positions at which  $P$   $p$ -matches within some  $3m/2$  length suffix of  $T$ . There exists a set  $Y$  with  $|Y| \leq 6|\Sigma|$  and an arithmetic progression  $\mathcal{A}$  such that  $Y \cup \mathcal{A} = X$ . Further,*

1. *all positions in  $Y$  are to the left of the leftmost position of  $\mathcal{A}$ ,*
2. *the distance between any two (consecutive) positions in  $\mathcal{A}$  is  $\rho$ , and*
3.  *$\text{pred}(T)[(i + m - \rho) \dots (i + m - 1)]$  is the same for all  $i \in \mathcal{A}$ .*

*Proof.* Since  $T$  is arbitrarily long and we are concerned with matches of the  $3m/2$  length suffix of  $T$ , we conceptually think of  $T$  as an array where all indices have been shifted by  $n - 3m/2$ . That is,  $T[-n + 3m/2]$  is the first character of  $T$ ,  $T[0]$  is the first character of the  $3m/2$  length suffix of  $T$  and  $T[3m/2 - 1]$  is the last character of  $T$ . We are now concerned with matches between  $P$  and  $T[0 \dots 3m/2 - 1]$ . We may of course translate back to the normal indexing by adding the offset  $n - 3m/2$ .

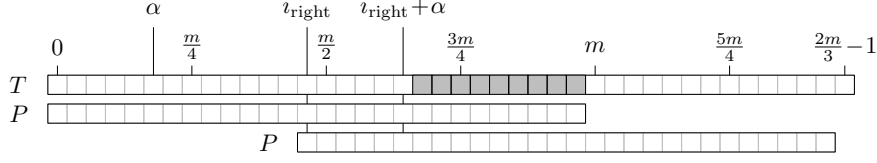
We may assume that  $P$  parameterise matches  $T$  at position 0 (i.e. the leftmost position of the  $3m/2$  length suffix of  $T$ ). Let  $\rho$  be the parameterised period of  $P$ . Let  $\alpha$  be the smallest number such that all distinct symbols in  $P$  occur in the prefix  $P[0 \dots \alpha]$ . By Lemma 11 we have that  $\rho \geq \alpha/|\Sigma|$ .

First consider the case  $\alpha \geq m/4$ . This implies that  $\rho \geq m/(4|\Sigma|)$  and the total number of positions where  $P$  can parameterise match  $T$  is upper bounded by

$$\frac{|T|}{\rho} = \frac{3m/2}{m/(4|\Sigma|)} \leq 6|\Sigma|.$$

All these positions can be stored in the set  $Y$ .

Now consider the case  $\alpha < m/4$ . Suppose first that  $\rho \geq m/8$ . The number of positions at which  $P$  can parameterise match  $T$  is then upper bounded by the



**Fig. 7.** A schematic example for the proof of Lemma 12.

constant 12. We therefore continue with the assumption that  $\rho < m/8$ . As  $\rho \geq \alpha/|\Sigma|$ , there are at most  $(\alpha + 1)/(\alpha/|\Sigma|) \leq 2|\Sigma|$  positions from the set  $\{0, \dots, \alpha\}$  at which  $P$  can parameterise match  $T$ . We can store these positions in the set  $Y$ . Next we will show that the positions from the set  $\{\alpha + 1, \dots, 3m/2 - 1\}$  at which  $P$  parameterise matches  $T$  can be represented by the arithmetic progression  $\mathcal{A}$ .

Since all distinct symbols in  $P$  occur in the prefix  $P[0 \dots \alpha]$  of  $P$ , it follows from Lemma 9 that  $\rho$  is an exact period of  $\text{pred}(P)[\alpha + 1 \dots m - 1]$  (not necessarily the shortest period). We have assumed that  $P$  parameterise matches  $T$  at position 0, so

$$\text{pred}(P) = \text{pred}(T[0 \dots m - 1]) = \text{pred}(T)[0 \dots m - 1],$$

which certainly implies that

$$\text{pred}(P)[\alpha + 1 \dots m - 1] = \text{pred}(T)[\alpha + 1 \dots m - 1].$$

Thus,  $\rho$  is also an exact period of  $\text{pred}(T)[\alpha + 1 \dots m - 1]$ .

Let  $i_{\text{right}}$  be the rightmost position at which  $P$  parameterise matches  $T$ . Since all distinct symbols in  $P$  occur in  $P[0 \dots \alpha]$  we have that

$$\text{pred}(P)[\alpha + 1 \dots m - 1] = \text{pred}(T)[i_{\text{right}} + \alpha + 1 \dots i_{\text{right}} + m - 1],$$

hence  $\rho$  is an exact period of  $\text{pred}(T)[i_{\text{right}} + \alpha + 1 \dots i_{\text{right}} + m - 1]$ .

For the remaining part of this proof it may help to consider the illustrative example in Figure 7. If  $i_{\text{right}} \leq \alpha$  then there are no positions to store in the arithmetic progression  $\mathcal{A}$  and we are done. Suppose therefore that  $i_{\text{right}} \geq \alpha$ . We must have  $i_{\text{right}} + m \leq 3m/2$ , which with  $\alpha < m/4$  implies that  $i_{\text{right}} + \alpha < 3m/4$ . Thus, if we let

$$\begin{aligned} S &= \text{pred}(T)[\alpha + 1 \dots i_{\text{right}} + m - 1], \\ S_{\text{pref}} &= \text{pred}(T)[\alpha + 1 \dots m - 1], \\ S_{\text{suf}} &= \text{pred}(T)[i_{\text{right}} + \alpha + 1 \dots i_{\text{right}} + m - 1], \end{aligned}$$

where  $S_{\text{pref}}$  is a prefix of  $S$  and  $S_{\text{suf}}$  is a suffix of  $S$ , we have that the length of the overlap of  $S_{\text{pref}}$  and  $S_{\text{suf}}$  (shaded section in Figure 7) is

$$(m - 1) - (i_{\text{right}} + \alpha + 1) + 1 > \frac{m}{4} - 1 > 2\rho - 1 \geq \rho.$$

We have shown above that both  $S_{\text{pref}}$  and  $S_{\text{suf}}$  have the exact period  $\rho$ , so by Lemma 10 it follows that  $\rho$  is also a period of  $S$ .

Let  $i$  be any position from the set  $\{\alpha + 1, \dots, i_{\text{right}} - 1\}$  such that  $P$  parameterise matches  $T$  at  $i$ . If no such  $i$  exists then  $i_{\text{right}}$  is the only position to store in the arithmetic progression  $\mathcal{A}$ . We will now show that if some  $i$  exists then  $P$  must also parameterise match  $T$  at position  $i + \rho$ . By induction, starting with the smallest  $i$ ,

the set  $\{i, (i + \rho), (i + 2\rho), \dots, i_{\text{right}}\}$  is the set of positions at which  $P$  parameterise matches  $T$ . This set is an arithmetic progression.

As  $\rho$  is an exact period of  $S$ , any two substrings of  $S$  that are  $\rho$  positions apart must be identical. Thus,

$$\text{pred}(T)[i \dots i + m - 1] = \text{pred}(T)[i + \rho \dots i + \rho + m - 1].$$

By Lemma 7 we have that

$$\text{pred}(T[i \dots i + m - 1]) = \text{pred}(T[i + \rho \dots i + \rho + m - 1]),$$

which implies that  $P$  parameterise matches  $T$  at position  $i + \rho$ .

Finally, the two properties in the statement of the lemma follow from the proof.  $\square$

*Proof (Proof of Lemma 6).* To take advantage of Lemma 12 we first conceptually partition the text into overlapping substrings  $T[k(m_{\ell-1}/2) \dots (k+3)(m_{\ell-1}/2) - 1]$  for all  $k \in \{0, 1, 2, \dots\}$ . Consider the set of p-matches for  $P_{\ell-1}$  that have been outputted by level  $\ell - 1$  but level  $\ell$  has not yet outputted  $\hat{\Phi}_{\ell}$ . As  $m_{\ell}/m_{\ell-1}$  is constant, at any time these p-matches are contained within a constant number of distinct partitions. By applying Lemma 12 using  $P_{\ell-1}$  as the pattern and we have that the p-matches in any partition form two sets  $Y$  and  $\mathcal{A}$ , where  $\mathcal{A}$  is an arithmetic progression. The set  $Y$  contains at most  $6|\Sigma|$  occurrences. By the first property in the statement of Lemma 12, the occurrences in  $Y$  all occur before the occurrences in  $\mathcal{A}$ . Therefore, we store  $\{\Phi_{\ell-1}(i') \mid i' \in Y\}$  explicitly (in left to right order) along with the text positions  $i'$  to which they correspond. This requires only  $O(|\Sigma|)$  space. Any additional occurrences which arrive must form an arithmetic progression. By the second and third properties in the statement of Lemma 12, we can store the set  $\{\Phi_{\ell-1}(i') \mid i' \in \mathcal{A}\}$  by storing only  $\Phi_{\ell-1}(i_1)$ ,  $\phi(\text{pred}(T)[(i_1 + m_{\ell-1} - \rho_{\ell-1}) \dots (i_1 + m_{\ell-1} - 1)])$  and the progression  $\mathcal{A}$  itself, where  $i_1$  is the first position in  $\mathcal{A}$  and  $\rho_{\ell-1}$  is the parameterised period of  $P_{\ell-1}$ . Any fingerprint in this set can be recovered in constant time as required by simple fingerprint arithmetic. We have therefore stored all the required fingerprints which occur in a single partition in  $O(|\Sigma|)$  space and as observed we only need to consider a constant number of partitions at one time to maintain  $M_{\ell}$ .